Projet IRGPU

Détection de mouvement

Calcul sur GPU

1 Introduction

Ce rapport présente notre implémentation d'un système de détection de mouvement en temps réel utilisant le GPU. L'objectif est de traiter un flux vidéo via GStreamer et d'identifier les zones en mouvement par soustraction de fond.

1.1 Contexte

La détection de mouvement est une tâche fondamentale en vision par ordinateur. Elle consiste à identifier les pixels qui diffèrent significativement d'un modèle de fond estimé. Cette opération est particulièrement adaptée au parallélisme massif offert par les GPU.

1.2 Pipeline de traitement

Notre pipeline se compose de 4 étapes principales :

1. **Estimation du fond** : Weighted Reservoir Sampling
2. **Masque de mouvement** : Différence RGB avec le fond
3. **Filtrage morphologique** : Ouverture (érosion + dilatation)
4. **Hystérésis** : Double seuillage avec reconstruction

2 Choix d'implémentation

2.1 Estimation du fond

Définition — Weighted Reservoir Sampling

Pour chaque pixel, on maintient $k = 4$ réservoirs contenant une couleur RGB et un poids. Lors de l'échantillonnage :

- Si un réservoir correspond (distance RGB < 30), on incrémente son poids
- Sinon, on remplace le réservoir de poids minimal avec probabilité inversement proportionnelle à son poids

Le fond est la couleur du réservoir de poids maximal.

Pourquoi ce choix ? Cette méthode s'adapte naturellement aux scènes avec plusieurs fonds alternés (ex : feuilles qui bougent) contrairement à une simple moyenne mobile.

2.2 Morphologie

L'ouverture morphologique (érosion puis dilatation) permet d'éliminer le bruit tout en préservant les grandes régions. Nous utilisons un **élément structurant en forme de disque** pour éviter les artefacts directionnels.

2.3 Hystérésis

Définition Double seuillage avec reconstruction

1. Les pixels avec score $> T_{high}$ sont marqués comme mouvement certain
2. Les pixels avec score $> T_{low}$ connectés à un pixel certain sont également marqués
3. Itération jusqu'à convergence

3 Implémentation GPU

3.1 Kernels CUDA

Tous les traitements sont implémentés en CUDA :

— `background_estimation_kernel` : Mise à jour des réservoirs avec cuRAND
— `motion_mask_kernel` : Calcul de la différence RGB
— `erosion_kernel` / `dilation_kernel` : Morphologie avec élément disque
— `hysteresis_reconstruction_kernel` : Propagation itérative avec flag atomique
— `visualization_kernel` : Overlay rouge sur les zones en mouvement

3.2 Optimisation clé : Hystérésis GPU

L'optimisation majeure a été le portage de l'hystérésis sur GPU. La version initiale effectuait :

1. Copie GPU → CPU
2. Boucle d'hystérésis sur CPU (jusqu'à 100 itérations)
3. Copie CPU → GPU

La version optimisée utilise un kernel itératif avec un flag atomique (`atomicOr`) pour détecter la convergence, éliminant les transferts mémoire coûteux.

4 Benchmarks

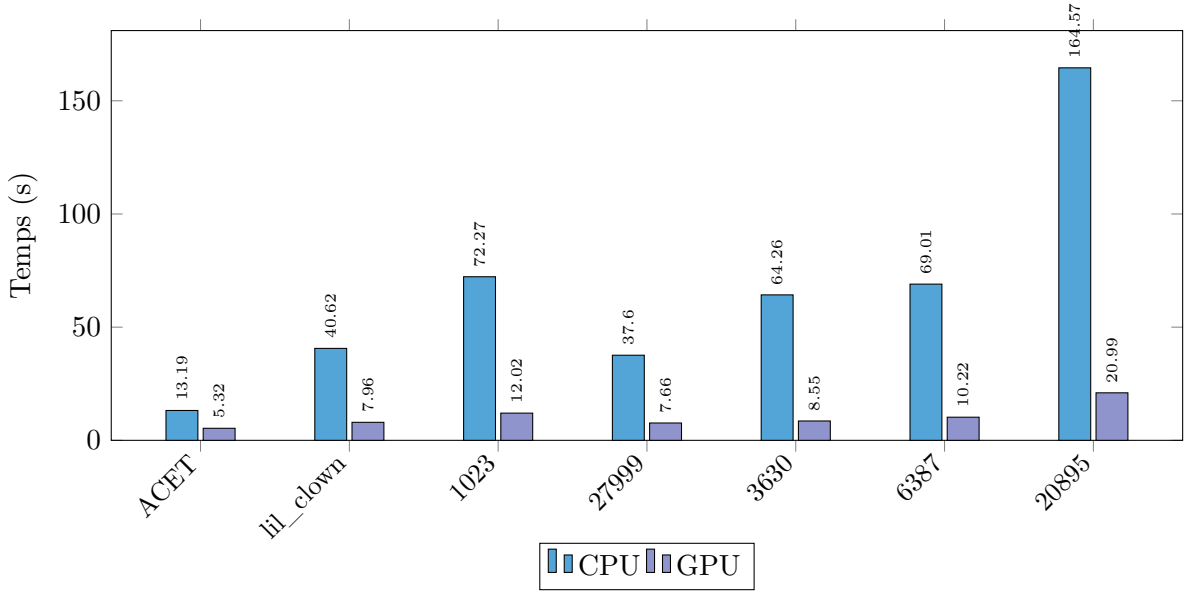
4.1 Configuration matérielle

GPU	NVIDIA GeForce RTX 5060
CPU	AMD Ryzen 9 5950X (16 cores, 32 threads)
RAM	58 Go

4.2 Résultats

Vidéo	CPU (s)	GPU (s)	Speedup
ACET.mp4	13.19	5.32	2.5×
lil_clown_studio.mp4	40.62	7.96	5.1×
1023-142621257_large.mp4	72.27	12.02	6.0×
27999-366978301_large.mp4	37.60	7.66	4.9×
3630-172488409_large.mp4	64.26	8.55	7.5×
6387-191695740_large.mp4	69.01	10.22	6.8×
20895-313083562_large.mp4	164.57	20.99	7.8×
Moyenne	-	-	**5.8**×

4.3 Graphique des performances



5 Analyse des performances

5.1 Impact des optimisations

Version	Optimisation	Impact
v1.0	Baseline (hystérésis CPU)	Référence
v1.1	Hystérésis full GPU	**+346%**
v1.2	Sans cudaDeviceSynchronize()	$\approx 0\%$
v1.3	Shared memory morphologie	$\approx 0\%$
v1.4	Fusion kernels	$\approx 0\%$

5.2 Bottlenecks identifiés

1. **Transferts Host↔Device** : Chaque frame est copiée vers le GPU puis le résultat est recopié. C'est le bottleneck principal après l'optimisation de l'hystérésis.
2. **Hystérésis itératif** : Le kernel d'hystérésis peut nécessiter jusqu'à 100 itérations pour converger, avec une synchronisation à chaque itération pour vérifier le flag.

5.3 Pistes d'amélioration

— **Zero-copy memory** : Utiliser la mémoire partagée CPU/GPU pour éviter les copies explicites

— **CUDA streams** : Overlapper les transferts et le calcul

— **Persistent kernels** : Garder les données sur le GPU entre les frames

6 Conclusion

Nous avons implémenté un système de détection de mouvement atteignant un speedup moyen de **5.8**× par rapport à la version CPU. L'optimisation clé a été le portage de l'hystérésis sur GPU, qui a permis d'éliminer les transferts mémoire coûteux entre les itérations.
Les optimisations classiques (shared memory, kernel fusion) n'ont pas apporté de gain significatif car le bottleneck se situe au niveau des transferts Host↔Device plutôt que dans le calcul lui-même.

Projet réalisé dans le cadre du cours GPGPU - EPITA ING3 - 2025